$\mathcal{F}$LORA-2: A Rule-Based Knowledge Representation and Inference Infrastructure for the Semantic Web*

Guizhen Yang[1], Michael Kifer[2], and Chang Zhao[2]

[1] Department of Computer Science and Engineering
University at Buffalo, Buffalo, NY 14260-2000, USA
gzyang@CSE.Buffalo.EDU

[2] Department of Computer Science
Stony Brook University, Stony Brook, NY 11794-4400, USA
{kifer,changz}@CS.StonyBrook.EDU

Abstract. $\mathcal{F}$LORA-2 is a rule-based object-oriented knowledge base system designed for a variety of automated tasks on the Semantic Web, ranging from meta-data management to information integration to intelligent agents. The $\mathcal{F}$LORA-2 system integrates F-logic, HiLog, and Transaction Logic into a coherent knowledge representation and inference language. The result is a flexible and natural framework that combines rule-based and object-oriented paradigms. This paper discusses the principles underlying the design of the $\mathcal{F}$LORA-2 system and describes its salient features, including meta-programming, reification, logical database updates, encapsulation, and support for dynamic modules.

1 Introduction

$\mathcal{F}$LORA-2 [42] is a rule-based knowledge representation and inference system, which seeks to provide a rich infrastructure for reasoning with semantic information on the Web. The logical foundations of $\mathcal{F}$LORA-2 are deeply rooted in F-logic [20], HiLog [9], and Transaction Logic [5]. Firstly, F-logic brings many important object-oriented features such as complex objects, class hierarchies, and inheritance. Secondly, HiLog provides a basis for reification and enhances F-logic with meta-information processing capabilities. Finally, Transaction Logic supports declarative programming of "procedural knowledge" that is often embedded in intelligent agents or Semantic Web services. In $\mathcal{F}$LORA-2 these three formalisms are seamlessly pieced together, thus providing a coherent framework for specifying and manipulating knowledge as well as meta-knowledge in a logically clean fashion.

In this paper we discuss the design principles underlying the $\mathcal{F}$LORA-2 system and present the main features of its language. In particular, we point out how these principles relate to the rule-based reasoning infrastructure for the Semantic Web and to the various features of the $\mathcal{F}$LORA-2 system. Of special interest are

* This work was supported in part by NSF grants IIS-0072927 and CCR-0311512.

many different extensions to F-logic, which have been developed in order to meet the specific requirements of the Semantic Web. These include a logical theory for nonmonotonic multiple inheritance [37]; reification [41], which extends the basic theory provided by HiLog; and support for anonymous resources [41].

Another important extension is the novel module system of $\mathcal{F}$LORA-2, which was designed explicitly with the goal of supporting intelligent agents and knowledge integration. Unlike other module systems, which are tied to specific program code, $\mathcal{F}$LORA-2 modules are abstractions that represent structural components of a running system. Program code can be loaded into any module on-the-fly, and the already loaded code can be replaced by other code while the system runs. New modules can also be created and loaded at run time. Finally, to accommodate a wide variety of semantic components in knowledge integration, different modules can have different (even customized) semantics.

This paper is organized as follows. Section 2 introduces the principles underlying the design of $\mathcal{F}$LORA-2. Section 3 provides an overview of the three logics underlying $\mathcal{F}$LORA-2 and presents the extensions that have been developed specifically to support the Semantic Web infrastructure. In Section 4, we introduce the module system of $\mathcal{F}$LORA-2. Section 5 concludes the paper with a discussion of future work.

2 Design Principles

In this section we discuss and motivate the main principles underlying the design of $\mathcal{F}$LORA-2. Some of the design decisions were adopted directly from the three logics: F-logic, HiLog, and Transaction Logic. Other principles stem from the high-level goal that motivated our effort — the development of a powerful rule-based language that is suitable for the Semantic Web.

2.1 Choice of Semantics

Asserted vs. Inferred Knowledge. While current standardization efforts, such as OWL [35], are focusing on first-order logic, new, forward-looking research is arguing that nonmonotonic extensions are needed for more sophisticated uses of the Semantic Web (*e.g.*, [1, 23]). In ontologies, nonmonotonicity arises in advanced applications, such as Semantic Web services; in agent-based systems, it arises in the form of common-sense reasoning.

In the past, claims has been made in various discussion forums that nonmonotonic closed-world assumption is inherently inappropriate for reasoning about the Web because the totality of all the information is not known in advance to the reasoner. We believe that this point of view stems from insufficient attention given to the view that knowledge on the Web comes in two main varieties: *asserted knowledge*, which is specified mostly as sets of facts, and *inferred knowledge*, which is derived using rules.

While the distinction between closed-world assumption and open-world assumption may seem insignificant for asserted knowledge, it makes a world of difference in the interpretation of rules. To date, the best studied and the most successful applications of rules in intelligent systems are based on a form of nonmonotonic closed-world assumption.

The misconception about the inapplicability of semantic closure originates, in part, in the tradition of logic programming, where data and rules are part of the same program. In contrast, the tradition of databases has always made a distinction between data and rules, and referred to these two kinds of knowledge as *extensional* and *intentional* database, respectively. The extensional database has been viewed as a variable part of the knowledge base, and closed-world inferences were always subject to change when the extensional part changes.

It is not hard to see that this point of view is very similar to the Web environment, where data sources can be viewed as sets of facts asserted into the extensional database. When data at the sources changes or the sources are added or deleted, the old nonmonotonic inferences are revised. For instance, an intelligent travel agent service can be defined by a set of $\mathcal{F}$LORA-2 rules, which are interpreted using the closed-world assumption. However, the concrete inferences made by the agent would depend on the specific data sources that the agent is aware of. Adding or removing these sources will change the recommendations made by the agent.

Comprehensive Semantics. An important principle underlying $\mathcal{F}$LORA-2 is that the language should be comprehensive and every syntactically correct specification should have a natural semantics. Since $\mathcal{F}$LORA-2 is as general as Prolog and thus is Turing-complete, it is possible to write non-terminating programs. In fact, since Transaction Logic is part of the language and thus $\mathcal{F}$LORA-2 programs can modify the underlying database, the language is Turing-complete even if we do not use function symbols in queries and rules [3]. However, it was always our intention to design a complete language that has tractable subsets suitable for query answering. For instance, [37] shows that query answering is polynomial in function-free $\mathcal{F}$LORA-2 programs even in the presence of nonmonotonic multiple inheritance, and [3] describes subsets of Transaction Logic with various degrees of computational complexity.

The semantics of a rule-based object-oriented language that supports multiple inheritance with overriding was known to be a hard issue. The original proposal [20], as well as a number of subsequent semantics [7, 17, 18, 29, 30], had suffered from a number of anomalies (see [39] for more details). A comprehensive solution was recently proposed in [39] and [40].

2.2 Lazy Assimilation of Knowledge

Due to the dynamic nature of data sources on the Web and general incompleteness of information in this medium, a Semantic Web language must be as impervious to changes in the asserted knowledge as possible.

Thus, Semantic Web applications need to be able to assimilate knowledge *lazily*, without having to *revise* existing knowledge substantially when new knowledge is added. This is especially important for data sources that are not under the control of client applications. When such sources change, they normally do not send notification to the client applications, and exhaustive querying of these sources (to determine what the changes are) is expensive and often impossible.

Partial Knowledge about Class Hierarchies and Objects. F-logic and $\mathcal{F}$LORA-2 address the above problems in several ways. First, subclass relationship, sub :: super, is not immediate, *i.e.*, it is not invalidated if a new intermediate class is discovered later on. Thus, sub :: intermediate and intermediate :: super do not contradict sub :: super. Likewise, class membership, object : class, is not immediate; it does not rule out the possibility that there may exist an intermediate class between object and class, *i.e.*, object : intermediate and intermediate :: class. This reflects the view that knowledge about class hierarchies is *incomplete*. Likewise, any assertion about an object is assumed to be incomplete. For instance, asserting that an object, say, john, has a set-valued attribute siblings with a known set of values, *e.g.*, {bill, mary}, does not imply that these are the only values. It will not be a contradiction if later we learn that anne is also a sibling. Moreover, the available knowledge of an object schema is also assumed to be incomplete (unless the user imposes a strict type constraint on the object). Thus, if john is said to be an object with the attributes name, parents, and siblings, adding information about a new attribute, say, address will remain consistent with the old knowledge about the schema. Note that traditional object-oriented programming and database languages take exactly the opposite view on each of the above four points.

Incomplete Knowledge about Identity. Logic-based languages do not normally support equality between terms and assume that distinct variable-free terms represent different entities. However, this assumption is not appropriate when applied to resources on the Semantic Web. For instance, different URIs might, in fact, represent the same resource. $\mathcal{F}$LORA-2 addresses this problem by offering an explicit equality operator, :=:, which has all the normal properties of equality: commutativity, transitivity, congruence, etc. Another source of incomplete identity comes from the so-called *anonymous resources* (or *blank nodes*) in RDF [26]. We discuss equality and anonymous resources in Section 3.

2.3 Integrated Meta-Knowledge Handling

Meta-Information. Most of Web data is semistructured in the sense that it has object-like structure with complex relationships among the objects, but the schema of these objects is not known to the application in advance. Two scenarios are possible here: (1) Data is shipped together with the schema. In this case the application must be able to query and explore the schema before processing the data. We call this scenario *schema-level meta-querying*; (2) Data is shipped without the schema, but it is self-describing (*i.e.*, some attribute information is embedded in the data). In this case, the application should be able to query whatever structural information is available as part of the data.

We call this situation *instance-level meta-querying*. We claim that querying meta-information should be as natural as querying instance data, and this capability should be integrated into the language *without* any extra machinery. The next section discusses how this is achieved in $\mathcal{F}$LORA-2 through its integration of F-logic and HiLog.

Reification. Since the advent of RDF [26], the ability to reify statements about Web resources has been viewed as one of the fundamental requirements for a Semantic Web language. Here, again, we claim that reification should be integrated into the language naturally, without extra machinery, and in a paradox-free way. Section 3.6 discusses how this is achieved in $\mathcal{F}$LORA-2 by building on the ideas underlying HiLog.

3 The $\mathcal{F}$LORA-2 Language

In this section we review the technical foundations of $\mathcal{F}$LORA-2 — F-logic [20], HiLog [9], and Transaction Logic [5] — and discuss various extensions to F-logic which were developed to meet the specific requirements of the Semantic Web.

3.1 F-logic

F-logic subsumes predicate calculus both syntactically and semantically by explicitly introducing the concepts adapted from object-oriented programming. At the same time, much of F-logic can be viewed as a syntactic variant of classical logic, which makes implementation through source-level translation possible. However, we will not discuss implementation issues here for want of space.

Basic Syntax. F-logic uses first-order variable-free terms to represent *object identity* (abbr., OID), *e.g.*, john and father(mary). Objects can have single-valued, set-valued or Boolean attributes, for instance,

mary[spouse → john, children ↠ {alice,nancy}].
mary[children ↠ {jack}, married].

These formulas are called F-logic *molecules*. Note that each formula above asserts several facts *simultaneously*. In the first formula spouse → john says that object mary has a single-valued attribute spouse, whose value is OID john, while children ↠ {alice, nancy} in the same object description says that the value of the set-valued attribute children is a set that *contains* two OIDs: alice and nancy. We emphasize "contains" because sets do not need to be specified all at once. For instance, the second formula above says that mary has another child jack. The attribute married in the second formula is Boolean: its value is *true* in the above example. This is one of the manifestations of the lazy knowledge assimilation principle of Section 2.

While some attributes of an object are specified explicitly, as facts, other attributes can be defined using inference rules. For instance, we can derive john[children ↠ {alice, nancy, jack}] using the following rule:

$\mathsf{X[children \twoheadrightarrow \{C\}] :- Y[spouse \rightarrow X,\ children \twoheadrightarrow \{C\}].}$

Here we adopt the standard convention that uppercase symbols denote variables while symbols beginning with a lowercase letter denote constants.

F-logic objects can also have *methods*, which are functions that take arguments. For instance,

$\mathsf{john[grade(cs305,f99) \rightarrow 100,\ courses(f99) \twoheadrightarrow \{cs305,cs306\}].}$

says that john has a single-valued method, grade, whose value on the arguments cs305 and f99 is 100; it also has a set-valued method courses, whose value on the argument f99 is a set of OIDs that contains cs305 and cs306. Like attributes, methods can be defined using inference rules.

The F-logic syntax for *class membership* is john : student and for *subclass relationship* it is student :: person. In addition, F-logic supports specification of schema information. For instance, $\mathsf{person[name \Rightarrow string,\ child \Rrightarrow person]}$ says that the signature of class person has two attributes, a single-valued attribute name and a set-valued attribute child. Moreover, the first attribute returns objects of type string and the second returns sets of objects such that each object in the set is of type person. F-logic also supports first-order predicate syntax and thus it integrates relational and object-oriented paradigms.

Querying Meta-Information. F-logic provides simple and natural facilities for exploring the structure of object data. Both schema information associated with classes and structures of individual objects can be queried by simply putting variables in the appropriate syntactic positions. For instance, to find out which set-valued methods defined in the *schema* of class student return objects of type person one can pose the following simple query:

?- $\mathsf{student[M \Rrightarrow person].}$

The following query returns the type of the attribute name in class student and all student's superclasses:

?- $\mathsf{student::C,\ student\,[name \Rightarrow T].}$

The above two queries involve subclass relationship and type information (as indicated by the operators ::, $\Rightarrow$, and $\Rrightarrow$); they are called *schema-level meta-queries.* In contrast the following two queries return all methods that are defined for the object with OID john[3]:

?- $\mathsf{john[SingleM \rightarrow \_].}$
?- $\mathsf{john[SetM \twoheadrightarrow \_].}$

Note that these queries check what is defined for the object itself rather than what is defined in the schema; they are thus called *instance-level meta-queries.* The two kinds of meta-queries can return different results for several reasons. First, in case of semistructured data, schema information might be incomplete

[3] Every occurrence of "_" stands for a don't-care variable.

and additional attributes can be defined for individual objects than what the schema reveals. Second, even if the schema is complete, the values of some attributes can be undefined for some of the objects in the class. In this latter case, the undefined attributes will not be returned by instance-level meta-queries, while they would be returned by schema-level meta-queries.

In Section 3.2, we will see more examples of meta-queries, which are enhanced by the facilities of HiLog.

Path Expressions. In addition to the basic syntax, F-logic supports *path expressions* to simplify navigation along single-valued and set-valued attribute and method invocations, and to avoid explicit join conditions [15]. The basic idea is to allow path expressions like $\mathsf{O.M}$ and $\mathsf{O..M}$ wherever OIDs are allowed.

A *single-valued* path expression, $\mathsf{O.M}$, refers to the *unique* object R for which $\mathsf{O[M \rightarrow R]}$ holds; a *set-valued* path expression, $\mathsf{O..M}$, refers to some object, R, such that $\mathsf{O[M \twoheadrightarrow \{R\}]}$ holds. Here the symbols O and M can be either an OID or a path expression. Furthermore, M can be a method with arguments, *e.g.*, $\mathsf{O.M(P_1,\ldots,P_k)}$ is a valid path expression that refers to the object R that satisfies $\mathsf{O[M(P_1,\ldots,P_k) \rightarrow R]}$.

Path expressions and F-logic formulas can be arbitrarily nested. This leads to a very concise and flexible query language for specifying object properties. For instance, the following path expression:

$$\mathsf{Paper[authors \twoheadrightarrow \{Author[name \rightarrow john]\}].publication..editors}$$

refers to all editors of those papers in which john is the name of a co-author. The reader has probably noticed the conceptual similarity of such extended path expressions with XPath, which was developed after the extended path expressions were introduced to F-logic in [15].

Equality. Unlike regular logic programming languages, such as Prolog, in F-logic variable-free terms can become *equal* because of single-valued attributes and methods. For instance, consider the following facts:

$$\mathsf{mary[spouse \rightarrow joseph].} \quad \mathsf{mary[spouse \rightarrow joe].} \quad \mathsf{joseph[son \twoheadrightarrow frank].}$$

Since spouse is a single-valued attribute, it can have at most one value for any given object. Therefore, the OIDs joseph and joe must refer to the same object and whatever is true about joseph should also be true about joe. In particular, we should be able to derive that $\mathsf{joe[son \twoheadrightarrow frank]}$.

On the Web, equality is also sometimes required to represent the fact that two different URIs represent the same resource. To accommodate this requirement, $\mathcal{F}$LORA-2 provides inference infrastructure for user-defined equality theories. This is made possible through the explicit equality predicate, :=:. Equality can be asserted as a fact or derived via rules, such as this:

$$\mathsf{U1 :=: U2 \ :- \ sameURL(U1,U2).}$$

$\mathcal{F}$LORA-2 supports two built-in equality theories: one where :=: is the usual congruent equivalence relationship and the other where the functional constraints imposed by single-valued methods are enforced, *i.e.*, $\mathsf{a[b \rightarrow c]}$ and $\mathsf{a[b \rightarrow d]}$ implies $\mathsf{c{:=:}d}$. The user can specify the desired default equality theory to use for each program module separately (see Section 4). The interested reader is referred to [42] for more information about equality maintenance in $\mathcal{F}$LORA-2.

3.2 HiLog

HiLog was introduced in [9] in order to extend logic programming with higher-order syntax, yet tractable and first-order semantics. In particular, the goal was to extend classical predicate calculus to enable flexible and natural querying of term structures and to support reification of atomic formulas. The simplest and yet most unusual illustration of HiLog is the following definition of the standard Prolog meta-predicate call:

$$\mathsf{call(X)} :- \mathsf{X}.$$

This means that HiLog does not distinguish between function terms and atomic formulas: The same variable can range over both and thus *atomic formulas are reified*. Variables can also range over function and predicate symbols, as in X(Y,a), and queries of the form ?– p(X), X, X(Y,X) are well within the boundaries of HiLog. The syntax for HiLog terms also extends that of classical logic. For instance, g(X)(f(a,X),Y)(b,Y) is perfectly fine, and there are several important uses for this multi-level syntax (see [9] for some).

Variables in the position of function and predicate symbols eliminate the need for many uses of non-logical meta-operators of Prolog and serve as a much more natural replacement for such uses. Combined with F-logic, HiLog enhances the already powerful meta-features of the language. For instance, in the combined language, one can write:

$$\mathsf{X[methods} \twoheadrightarrow \{\mathsf{M}\}] :- \mathsf{X[M(\_,\_)} \rightarrow \_].$$

Thus, a query of the form

$$?\text{-}\ \mathsf{john[methods} \twoheadrightarrow \{\mathsf{M}\}].$$

will return the set of all 2-argument set-valued methods defined for the object john, while the query

$$\mathsf{X[methods} \twoheadrightarrow \{\mathsf{M}\}] :- \mathsf{X[M(\_)} \Rightarrow \_].$$
$$?\text{-}\ \mathsf{student[methods} \twoheadrightarrow \{\mathsf{M}\}].$$

returns the set of all single-valued methods defined in the schema (signature) of class student.

3.3 Transaction Logic

A programming language, especially an object-oriented programming language, needs primitives for modifying the underlying state of the system. In a logic-based language, modifying the underlying state means updating the database part of the program. This need was recognized by the designers of Prolog who introduced the well-known assert and retract operators. From the very beginning, assert and retract were perceived as the necessary evil in the absence of a truly logical solution. Various attempts at formalizing updates in a logic programming language met with limited success (*e.g.*, [24, 33, 27]). A detailed discussion of this subject appears in [5]. One of the most serious drawbacks of these approaches is that they impose special programming styles (which require significant programming effort) and that they do not support subroutines — one of the most fundamental aspects of any programming language.

Transaction Logic [4, 5] provides a comprehensive theory of logical updates in logic programming, which does not suffer from any of the above drawbacks, and its programming style is very much in the spirit of Prolog. The utility of Transaction Logic has been demonstrated on a vast range of applications: from databases to robot action planning to reasoning about actions to workflow analysis [6, 10].

One of the implications of the update semantics provided by Transaction Logic is that update transactions are *atomic*. For instance, in Prolog, if a post-condition of a state-changing operator is false, the entire execution fails, returning the answer "No". Despite that, all the changes made by assert and retract would stay and the database may be left in a inconsistent state. This non-logical property is responsible for many complications in Prolog programming. Transaction Logic rectifies this and similar problems with updates in logic programming.

$\mathcal{F}$LORA-2 integrates F-logic and Transaction Logic along the lines of [19] with new extensions, which distinguishes queries from transactions and thus enables a number of compile-time checks. In Transaction Logic, both actions (transactions) and queries are represented as predicates. In $\mathcal{F}$LORA-2, transactions are expressed as object methods that are prefixed with the special symbol "#".

The following program is an implementation of a block-stacking robot in $\mathcal{F}$LORA-2. Here, the action stack is defined as a Boolean method of the robot.

```
R[#stack(0,X)]   :-  R:robot.
R[#stack(N,X)]   :-  R:robot, N > 0,
                     Y[#move(X)], R[#stack(N-1,Y)].
Y[#move(X)]      :-  Y:block, Y[clear], X[clear], X[widerThen(Y)],
                     delete{Y[on → Z]}, insert{Z[clear]},
                     insert{Y[on → X]}, delete{X[clear]}.
```

Informally, the program says that to stack a pyramid of N blocks on top of block X, the robot must find a block Y, move it onto X, and then stack N-1 blocks on top of Y. To move Y onto X, both blocks must be "clear" (*i.e.*, with no other block sitting on top of them), and X must be wider than Y. If these conditions are satisfied, the database will be updated accordingly. If any of the conditions fails, it means that the current attempted execution is not a valid try and another attempt will be made. If no valid execution is found, the transaction fails and no changes will be made to the database. Again, we would like to point out that in a similar situation an analogous Prolog program can leave the database in an inconsistent state.

3.4 Value and Code Inheritance

Object-oriented languages normally distinguish between *instance methods* and *class methods*. The former characterize all instances of a class while the latter characterize classes as objects [32]. Class methods are analogous to "static" methods in Java and instance methods correspond to nonstatic methods (which are sometimes also called "instance methods" in Java). In object-oriented data modeling, especially in dealing with semistructured objects on the Web, it is also useful to consider *object methods* that are defined explicitly for individual objects and override inheritance from superclasses. Object methods are similar to class methods except that they cannot be inherited. All three kinds of methods are specified using rules; the differences among them are illustrated with the following examples.

Example 1. Suppose we want to compute bonus for employees in the software department. Our policy is to award bonus based on the overall sales of the entire department. For example, every employee gets a bonus of 1% of the total amount of sales. This policy can be represented as follows:

```
softDept[bonus ->> N]  :- softDept[salesTotal ->> S], N is S * 1%.
softDept[salesTotal ->> 1000].
john : softDept.
mary : softDept.
```

The first two clauses in the above program are class method definitions for the methods bonus and salesTotal, respectively. The first rule defines the method bonus, whose value depends on the class method salesTotal, whose value is specified in the second clause. Both methods are class methods in the class salesTotal. With these rules, we can infer softDept[bonus $\twoheadrightarrow$ 10].

The last two facts simply state that john and mary are members of the class softDept. Although the program does not explicitly define the method bonus for john, since john is a member of the class softDept, it will inherit the method bonus together with its value (*i.e.*, 10). Similarly, we can derive mary[bonus $\twoheadrightarrow$ 10].

Example 2. Rather than giving the same bonus to every employee, suppose that the hardware department has a policy that rewards individual performance, where an employee gets a bonus of 10% of the amount of his/her sales. This idea can be illustrated using the following program:

```
code hardDept[bonus ->> N]  :- hardDept[sales ->> S], N is S * 10%.
     mike : hardDept.
     lucy : hardDept.
     mike[sales ->> 300].
     lucy[sales ->> 200].
```

Note that the first rule is preceded with a special keyword, code, which marks the definition of an *instance method*, bonus, for class hardDept. Recall that instance methods, as part of class specifications, apply to every member of the class. Intuitively, the symbol hardDept in the above rule is treated as a "placeholder" that stands for every member of the class hardDept. The remaining clauses state that mike and lucy are members of hardDept, and provide sales figures for mike and lucy, respectively.

Let us examine how the method bonus is computed for mike. Since mike is a member of hardDept and the first rule in the last program defines the method bonus for all instances of hardDept, mike inherits the following instantiated rule:

$$\mathsf{mike[bonus} \twoheadrightarrow \mathsf{N]} \ :- \ \mathsf{mike[sales} \twoheadrightarrow \mathsf{S], N\ is\ S} * \mathsf{10\%.}$$

where mike is substituted for hardDept. This instantiation corresponds to the so called *late binding* in traditional object-oriented languages like Java. From this rule and the facts, we can derive $\mathsf{mike[bonus} \twoheadrightarrow \mathsf{30]}$ and $\mathsf{lucy[bonus} \twoheadrightarrow \mathsf{20]}$.

Inheritance via instance method definitions, as illustrated in Example 2, is called *code inheritance*, because what is inherited is the code rather than the result returned by the method (as in Example 1). Code inheritance is commonly used in imperative object-oriented languages such as C^{++} and Java. Inheritance via class method definitions, as illustrated in Example 1, is called *value inheritance*, because what is inherited is the result returned by the method. This kind of inheritance is commonly used in AI [36, 25].

The original F-logic [20] supports value inheritance only (and not satisfactorily at that — see [39]). Since $\mathcal{F}$LORA-2 is a practical knowledge engineering environment, it requires a theory of code inheritance as well. The interaction of the two kinds of inheritance introduces a number of interesting semantic and algorithmic problems, which were solved in [37].

3.5 Anonymous Identity

It has been argued in [11] that F-logic is a natural formalism to provide inference service for RDF(S) [26]. Representation of RDF statements with named resources in F-logic is rather straightforward. For instance, the statement "*Thomas Edison is the inventor of the bulb (denoted by URI http://foo.org/TheBulb)*" directly corresponds to the following F-logic statement

$$\mathsf{'http://foo.org/TheBulb'[inventor} \twoheadrightarrow \mathsf{'Thomas\ Edison'].}$$

However, one difficulty arises in representing RDF statements with *anonymous resources*. Consider the following statement: "*Someone, named Thomas Edison, born in 1847, is the inventor of the resource http://foo.org/TheBulb.*" The intent here is to make a structured resource *without a known object ID* and assert that it has two properties, name and born.

Anonymous resources were not envisioned in the original work on F-logic [20], but appropriate extensions were introduced in $\mathcal{F}$LORA-2 [42, 38]. To represent anonymous objects, $\mathcal{F}$LORA-2 uses a special symbol, _#, called an *unnumbered anonymous ID symbol*, and a countable set of *numbered anonymous ID symbols*: _#1, _#2, ..., etc. The intended meaning is that each occurrence of _# denotes a distinct object ID that does not occur anywhere else in the program. All occurrences of the same numbered anonymous ID symbol, *e.g.*, _#1, within the same clause are treated as representing the same object ID, but this ID is distinct from any other ID used elsewhere in the program (including the occurrences of _#1 in a different clause). The reader is referred to [41] for a formal treatment of anonymous ID symbols in $\mathcal{F}$LORA-2.

Thus, in $\mathcal{F}$LORA-2, the above statement can be represented as follows:

```
'http://foo.org/TheBulb'[inventor ->> _#1],
_#1[name ->> 'Thomas Edison', born ->> 1847].
```

Note that here the two occurrences of _#1 are within the same clause (here "," means conjunction) and thus refer to the same object. If we want to state that "*Someone invented the bulb and someone called Thomas Edison was born in 1847*", then we could write

```
'http://foo.org/TheBulb'[inventor ->> _#],
_#[name ->> 'Thomas Edison', born ->> 1847].
```

Here we use unnumbered anonymous ID symbols. Even though they occur within the same clause, they refer to distinct objects.

The semantics of anonymous ID symbols in $\mathcal{F}$LORA-2 as well as other uses of this extension (*e.g.*, to represent RDF *containers*) are described in [41].

3.6 Reification

Reification is needed to make statements about statements and is considered an important part of RDF. Since statements are formulas, making statements about them implies that formulas must be somehow treated as objects.

In $\mathcal{F}$LORA-2, reification is specified using a new language construct, ${...}. The statement inside ${...} is reified and this reified formula itself is treated as an object identity. For instance, the statement "*Someone named John Doe believes that a person, called Thomas Edison, invented the bulb (represented by the resource* http://foo.org/TheBulb*)*" is expressed in $\mathcal{F}$LORA-2 as follows:

```
_#[
  name ->> 'John Doe',
  believes ->>
    ${'http://foo.org/TheBulb'[inventor ->> {_#[name ->> 'Thomas Edison]}]}
].
```

In $\mathcal{F}$LORA-2, one can reason about reified statements in many interesting ways. For example, conjunctions, disjunctions, and even negations of formulas can be reified. Because reification can lead to logical paradoxes, especially in an expressive language like $\mathcal{F}$LORA-2, restrictions must be imposed on the language to prevent paradoxes. It turns out that it is enough to prohibit rules of the form X :− body, where X is a variable. A detailed account of the semantics of reification and of how paradoxes are avoided in $\mathcal{F}$LORA-2 can be found in [41].

4 Modules and Knowledge Integration

One of the most interesting and novel aspects of $\mathcal{F}$LORA-2 is its module system. It was designed with the intent to support autonomous agents and knowledge integration.

A $\mathcal{F}$LORA-2 module is an abstract container for a collection of data (which can be in main memory, a file, or a database), a chunk of program code, or an external procedure written in a different language. Modules can be created on the fly, and code or data can be loaded into the modules at runtime. Modules can be encapsulated, and the user can even specify the semantics for making inferences inside any module.

This section provides further details of the $\mathcal{F}$LORA-2 module system and discusses some applications.

Dynamic Module System. A $\mathcal{F}$LORA-2 program may consist of multiple modules. A *module* has a *name*, which other modules use to refer to information in that module, and the *content*. The content can be defined when the module is created, or it can be loaded (and, in fact, replaced) at any moment during runtime. The content of a module normally contains definitions of several classes, predicates, objects, and rules. Rules and data loaded into different modules do not interfere with each other, but they can interact: rule bodies can contain queries and even update operators that refer to other modules. This interaction is governed by encapsulation policies to be discussed later.

Suppose our program includes module people, which defines classes person, student, etc. Assume that student-objects expose a single-valued attribute name and a set-valued attribute major. Let us further assume that these definitions and rules are stored in a file named foo.flr. We can load this file into module people, and other modules will be able to query student-objects in it as follows:

```
?- load(foo >> people).
?- S:student[name → Name, major ↠ 'Computer Science']@people.
```

One can load a different file, say, bar.flr, into the same module at any time during execution with the command load(bar >> people). Since other modules have to know how to query the objects in module people, the code in file bar.flr would normally export the same interface to other modules. However, this is not always necessary, due to the flexible meta-programming features of $\mathcal{F}$LORA-2. A client module can *discover* the methods exported by the new content of people through a series of simple queries, such as

```
?- _[SingValMeth → _]@people, _[SetValMeth ↠ _]@people.
```

where "_" is an anonymous variable. It is even possible to find out which modules define, say, attribute major, by placing a variable in the module position:

```
?- _[major ->> _]@Module.
```

This type of flexibility is important in knowledge integration systems where the mediators can formulate the right queries to the various data sources based on the results of previously posed meta-queries.

A module can also be created empty and then filled in with rules and data. For instance, the statement

```
?- newmodule{stuff}.
```

creates a module named stuff; it can be filled with content either by loading it from a file (using the load statement) or through insertion of facts and rules at runtime:

```
?- insert{p(a)@stuff, john[major ->> cs]@stuff}.
?- insertrule{(q(X):-p(X))@stuff, (X[Y -> Z] :- p(X,Z),q(Y))@stuff}.
```

The content of a module, say stuff, can be manipulated freely by the *owner* module — the module that created module stuff (*i.e.*, the one that executed the newmodule command). Previously inserted facts and rules can also be deleted.

```
?- delete{p(a)@stuff}.
?- deleterule{(q(X):-p(X))@stuff}.
```

Such ability to create the content of a module on-the-fly is very useful for systems of autonomous intelligent agents. An agent may need to create a new module dynamically for several reasons. One is to store acquired data or rules, which constitute a separate corpus of knowledge. Storing this corpus in a separate module can prevent unforeseen interaction with the agent's main code or other acquired knowledge. Second, an agent may need to create another module in order to spawn off a "child" agent, similarly to how processes create child processes in operating systems.[4]

Encapsulation. Encapsulation is a generally accepted software engineering device, which helps prevent erroneous interactions among software components. $\mathcal{F}$LORA-2 supports encapsulation at the level of modules, and each module can encapsulate several classes. To limit the ways in which other modules can interact with a given module, the latter exports the methods and predicates that other modules can query or update. This is accomplished using the export directive. When it is used as a compile-time directive, the exported interfaces become available when the program is loaded into a module. An export instruction can also be executed at runtime. A method or a predicate can be exported as either readable or updatable. The latter allows the corresponding facts to be both queried and modified. Finally, interfaces can be exported to all modules or only to a specific list of modules.

The following directive, if executed in a module, people, will allow every other module to query the membership of class student and the membership of each of its subclasses.

[4] A module designer or owner can also protect the content from being owned (and thus modified) by other modules through the noowner directive.

```
?- export _:student readable.
```

In contrast, the following directive

```
?- export _[major →→ _] updatable to administrator.
```

will allow the administrator module (but no other modules) to not only query, but also change the majors of student-objects in module person. If module root is the owner of module people then it can also control what is exported by module person by executing an appropriate export directive in the latter module. For instance, executing

```
?- (export _:major readable)@person.
```

will give the right to query student majors, but the update privilege stays with the administrator module only (and the owner module).

Customized Semantics of Modules. A system that integrates different information sources or agents should be prepared to deal with modules that are implemented using different semantics. The module system of $\mathcal{F}$LORA-2 has been specially designed so as to allow each module to have its own semantics, which can vary according to three parameters: equality maintenance, inheritance semantics, and customized semantics.

– *Equality Maintenance.* As discussed in Sections 2 and 3.1, it is important to be able to equate OIDs that denote the same resource. However, $\mathcal{F}$LORA-2 recognizes that different equality theories are possible. First, there is a standard theory, where equality is a congruent equivalence relationship. F-logic [20] introduces one additional axiom that allows to infer new equalities from single-valued methods. Finally, for efficiency reasons, it is important to be able to tell the system that no equality maintenance is required if a certain module is known to not need this feature. Thus, at present, $\mathcal{F}$LORA-2 supports three equality options: basic, flogic, and none.
– *Inheritance Semantics.* This is the second parameter that a knowledge engineer can vary to tune the semantics of a module. At present, the user can request either the default inheritance semantics, as described in [39] (using the flogic option), or inheritance can be turned off (which can speed up queries, if inheritance is not needed).
– *Customized Semantics.* $\mathcal{F}$LORA-2 provides APIs, which allow the user to specify the semantics of a module through a set of axioms, which will be loaded into the module when it is created. The set of the axioms is placed in a file, which is communicated to the module loader using the custom option.

For instance, the following directive says that the module should be created with no support for equality, with F-logic style inheritance, and additional axioms that are defined in a file:

```
:- setsemantics equality(none), inheritance(flogic), custom('myaxioms.P').
```

The system also provides a primitive to query the semantic parameters used by any of the currently loaded modules. Thus, the architecture of $\mathcal{F}$LORA-2 supports integration of heterogeneous resources with different semantics.

5 Discussion and Conclusion

$\mathcal{F}$LORA-2 provides a large number of fundamental features that are essential for modeling semantic information on the Web. These include rules, classification, and frame-based syntax, powerful ways of processing meta-information, reification, transactional updates, and so on. All this is achieved using a natural and coherent syntax, which is supported by a comprehensive model-theoretic semantics for the unified language. In this way, $\mathcal{F}$LORA-2 is an embodiment of the principles outlined in Section 2.

We should note that other Semantic Web languages, such as OWL [35] and RDF [26], do follow some of the principles discussed in Section 2. However, these languages are rather limited in their features compared with $\mathcal{F}$LORA-2. In particular, they do not support general rules (including recursive rules and rules with negation in the rule body), common-sense reasoning, multiple inheritance, or logical updates. As a result, designing such languages and defining their semantics is in many ways a simpler task.

$\mathcal{F}$LORA-2 was inspired by and has inspired a number of other F-logic based systems, such as FLORID [14], TFL [8], FLIP [28], Ontobroker [12], and TRIPLE [34]. None of these systems supports HiLog, none (except TFL) supports Transaction Logic, none (except FLORID) supports inheritance, and none (except TRIPLE) supports reification.

The module systems of $\mathcal{F}$LORA-2 has some similarity with TRIPLE. However, in TRIPLE modules cannot be created and reloaded at run time — they have to be defined before the system starts running. Likewise, TRIPLE does not support encapsulation and, although one of its design goals is to enable integration of modules with different semantics, the system does not provide infrastructural support for this goal.

We have presented an overview of the $\mathcal{F}$LORA-2 system, its underlying design principles, logical foundations, language features, as well as its novel module system. Although $\mathcal{F}$LORA-2 already provides a wealth of features in support of semantic reasoning on the Web, a number of important issues still need to be addressed. For instance, we believe that the ability to handle inconsistent information should be part of the infrastructure. Handling imprecise and probabilistic information is another important missing piece. These issues can possibly be addressed with the help of paraconsistent and probabilistic logics such as [2, 21, 22, 31]. Another possible extension could be in the direction of prioritized rules, such as those described in [16].

References

1. A. Bernstein and B. N. Grosof. Beyond monotonic inheritance: Towards semantic web process ontologies. unpublished manuscript, 2003.
2. H. Blair and V. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science*, 68:135–154, 1989.
3. A. J. Bonner. Workflow, transactions, and datalog. In *ACM International Symposium on Principles of Database Systems (PODS)*, 1999.

4. A. J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133:205–265, October 1994.
5. A. J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers, March 1998.
6. A. J. Bonner and M. Kifer. Results on reasoning about action in transaction logic. In [13]. Springer-Verlag, 1998.
7. M. Bugliesi and H. M. Jamil. A stable model semantics for behavioral inheritance in deductive object oriented languages. In *International Conference on Database Theory (ICDT)*, 1995.
8. J. Carsi, P. Letelier, and P. Palma. A dood system for treating the schema evolution problem, 1998.
9. W. Chen, M. Kifer, and D. S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.
10. H. Davulcu, M. Kifer, C. Ramakrishnan, and I. Ramakrishnan. Logic based modeling and analysis of workflows. In *ACM International Symposium on Principles of Database Systems (PODS)*, 1998.
11. S. Decker, D. Brickley, J. Saarela, and J. Angele. A query and inference service for RDF. In *QL'98 - The Query Languages Workshop*, December 1998.
12. S. Decker, M. Erdmann, D. Fensel, and R. Studer. Ontobroker: Ontology based access to distributed and semi-structured information. In R. M. et al., editor, *Database Semantics, Semantic Issues in Multimedia Systems*, pages 351–369. Kluwer Academic Publisher, Boston, 1999.
13. B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors. *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*. Springer-Verlag, Berlin, 1998.
14. J. Frohn, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. Managing semistructured data with FLORID: A deductive object-oriented perspective. *Information Systems*, 23(8):589–613, 1998.
15. J. Frohn, G. Lausen, and H. Uphoff. Access to objects by path expressions and rules. In *International Conference on Very Large Data Bases (VLDB)*, 1994.
16. B. N. Grosof. Prioritized conflict handling for logic programs. In *International Logic Programming Symposium*, 1997.
17. H. M. Jamil. Implementing abstract objects with inheritance in Datalog$^{\text{neg}}$. In *International Conference on Very Large Data Bases (VLDB)*, 1997.
18. H. M. Jamil. A logic-based language for parametric inheritance. In A. G. Cohn, F. Giunchiglia, and B. Selman, editors, *KR2000: Principles of Knowledge Representation and Reasoning*, San Francisco, 2000. Morgan Kaufmann.
19. M. Kifer. Deductive and object-oriented data languages: A quest for integration. In *International Conference on Deductive and Object-Oriented Databases (DOOD)*, volume 1013 of *Lecture Notes in Computer Science*, Singapore, 1995. Springer-Verlag. Keynote Address at the 3rd International Conference on Deductive and Object-Oriented databases.
20. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM (JACM)*, 42:741–843, July 1995.
21. M. Kifer and E. Lozinskii. A logic for reasoning with inconsistency. *Journal of Automated Reasoning*, 9(2):179–215, November 1992.
22. M. Kifer and V. Subrahmanian. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12(4):335–368, April 1992.
23. H.-G. Kim. Pragmatics of the semantic web. In *Semantic Web Workshop at WWW-2002*, 2002.

24. R. A. Kowalski. Database updates in event calculus. *Journal of Logic Programming*, 12(1&2):121–146, January 1992.
25. L. V. S. Lakshmanan and K. Thirunarayan. Declarative frameworks for inheritance. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 357–388. Kluwer Academic Publishers, 1998.
26. O. Lasilla and R. S. (editors). Resource description framework (RDF) model and syntax specification. Technical report, W3C, February 1999. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.
27. G. Lausen and B. Ludäscher. Updates by reasoning about states. In *Second International East/West Database Workshop*, Klagenfurt, Austria, September 1994.
28. B. Ludäscher. The FLIP system (F-logic to XSB-Prolog compiler). http://www.informatik.uni-freiburg.de/˜ludaesch/flip/, 1994.
29. W. May and P. Kandzia. Nonmonotonic inheritance in object-oriented deductive database languages. *Journal of Logic and Computation*, 11(4), 2001.
30. W. May, B. Ludäscher, and G. Lausen. Well-founded semantics for deductive object-oriented database languages. In *International Conference on Deductive and Object-Oriented Databases (DOOD)*, 1997.
31. R. Ng and V. Subrahmanian. Probabilistic logic programming. *Information and Computation*, 101(2):150–201, December 1992.
32. F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems*, 16(1):88–131, March 1991.
33. R. Reiter. Formalizing database evolution in the situation calculus. In *Conference on Fifth Generation Computer Systems*, 1992.
34. M. Sintek and S. Decker. TRIPLE – a query, inference, and transformation language for the semantic web. In *International Semantic Web Conference*, 2002.
35. M. K. Smith, C. Welty, and D. L. McGuinness. OWL web ontology language guide. http://www.w3.org/TR/owl-guide/, 2003.
36. D. S. Touretzky. *The Mathematics of Inheritance*. Morgan-Kaufmann, Los Altos, CA, 1986.
37. G. Yang. *A Model Theory for Nonmonotonic Multiple Value and Code Inheritance in Object-Oriented Knowledge Bases*. PhD thesis, SUNY at Stony Brook, December 2002.
38. G. Yang and M. Kifer. Implementing an efficient DOOD system using a tabling logic engine. In *First International Conference on Computational Logic, DOOD'2000 Stream*, July 2000.
39. G. Yang and M. Kifer. Well-founded optimism: Inheritance in frame-based knowledge bases. In *International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE)*, October 2002.
40. G. Yang and M. Kifer. Inheritance and rules in object-oriented semantic web languages. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML)*, 2003.
41. G. Yang and M. Kifer. Reasoning about anonymous resources and meta statements on the semantic web. *Journal of Data Semantics*, 2004. To appear.
42. G. Yang, M. Kifer, and C. Zhao. $\mathcal{F}$LORA-2: User's Manual. http://flora.sourceforge.net/, June 2002.